Fidelity LEAP

Technology Immersion Program

Working with Relational Databases

Chapter 7: Databases with JDBC (Java Database Connectivity)

Chapter Overview

In this chapter, we will explore:

- How Java provides a set of interfaces for connecting to and working with SQL databases
  - Database vendors provide implementations of those interfaces
  - Allows you to write reusable database code
- Why working with databases requires a standard set of steps
  - You will practice these steps and then build them into an object
- Why database security is essential
  - Guard against bad user input
  - SQL injection attacks
- Data Access Object Design Pattern
  - What is the problem we need to solve?
  - What does the implementation look like?
  - What are the trade-offs?

Chapter Concepts

JDBC

Executing Queries

Implementing a Data Access Object

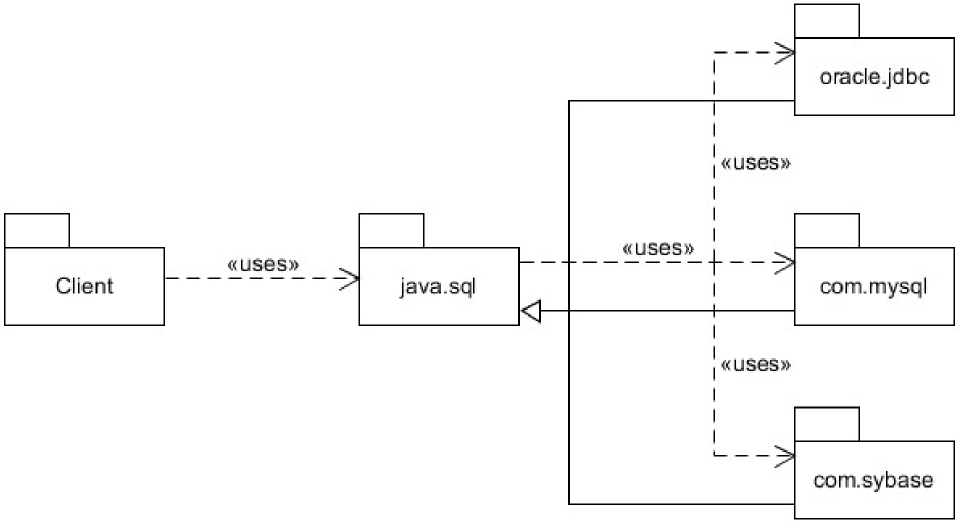
Handling NULL

Enumerated Types

Chapter Summary

`java.sql`

- Java provides a set of **interfaces** that offer a portable means of accessing databases
  - Java Database Connectivity (JDBC), supplied by `java.sql`
  - Supports standard SQL-92 syntax
  - The same Java code can access Oracle, MySQL, or Sybase database
    - The database vendors provide *drivers* that hook into `java.sql`



Connecting and Executing Queries

- To connect to a database and execute queries, every application has to:
  1. Load the database driver (only required for **very** old versions of Java)
  2. Create a Connection to the database
  3. Create a statement to execute SQL queries
  4. Parse the returned results of database call
  5. Close results and statements
  6. Close Connection

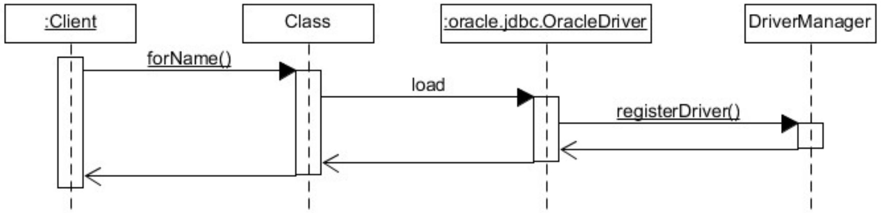
- Errors can occur at **any** of these steps
  - SqlExceptions will be thrown when an error happens

1. Loading and Registering the Driver

- **If using JBDC 4 (Java 6), or beyond, this is not needed:**
  - Any driver visible to the JVM at start-up is automatically loaded
- The client code has to load up the database vendor's driver

```
Class.forName("oracle.jdbc.OracleDriver");
```

  - The driver will then register itself with java.sql.DriverManager



  - Only needs to be done once
- If using MySQL, would register MySQL driver instead:

```
Class.forName("com.mysql.jdbc.Driver");
```

2. Creating Connection to Database

- The `DriverManager` can create a Connection to the database//BRAD NOTE for lab ☺

```
public Connection getConnection() {
  String dbUrl = "jdbc:oracle:thin@roifmrwinvm:1521/XE";
  String user = "scott";
  String password = "TIGER";
  Connection conn = null;

  try {
    conn = DriverManager.getConnection(dbUrl, user, password);
  } catch (SQLException e) {
     e.printStackTrace(); // better way coming soon
  }

  return conn;
}
```

Use the same settings as you used in SQL Developer

Password is case sensitive

```
// For MySQL
String dbUrl = "jdbc:mysql://localhost/mydb";
String user = "root";
String password = "root";
```

Reading Database Connection Properties

- It is possible to read the database Connection properties from a file//BRAD NOTE is try-catch needed?
YES!!!

```
Properties properties = new Properties();
properties.load(this.getClass()
                    .getClassLoader()
                    .getResourceAsStream("db.properties"));

String dbUrl    = properties.getProperty("db.url");
String user     = properties.getProperty("db.username");
String password = properties.getProperty("db.password");
```

db.properties

```
db.url=jdbc:oracle:thin:@localhost:1521/XEPDB1
db.username=scott
db.password=TIGER
db.driver=oracle.jdbc.driver.OracleDriver
```

DataSource

- Opening and closing database Connections is expensive
  – It is more efficient to reuse Connections to the same database
- Many JEE application servers provide an implementation of `javax.sql.DataSource`
  – Most provide a pool of database Connections
  – Requires a Java Naming and Directory (JNDI) service
    ▪ Provided by the application server
  – It is possible to request Connections capable of participating in distributed transactions
- We will use a very simple DataSource for testing our JDBC code
  – This does NOT use a Connection pool

```
DataSource dataSource = new SimpleDataSource();
Connection connection = dataSource.getConnection();
```

Exercise 7.1: Connecting to a Database

Do Now

20 min

- Complete this exercise described in the Exercise Manual
- Use TDD—verify your code works (i.e., does not throw an exception)
- The dependency for the Oracle database must be in `pom.xml`

```
<dependency>
  <groupId>com.oracle.database.jdbc</groupId>
  <artifactId>ojdbc8</artifactId>
  <version>18.3.0.0</version>
</dependency>
```

Chapter Concepts

JDBC

Executing Queries

Implementing a Data Access Object

Handling NULL

Enumerated Types

Chapter Summary

JDBC Statements

- The Connection can create a Statement to execute an SQL command
- There are three types of JDBC Statements:

1. Statement
   - Don’t use this! ☠
   - Vulnerable to SQL Injection exploits
2. PreparedStatement
   - Always use this for SQL commands
3. CallableStatement
   - Use this for executing a stored procedure in the database

User Inputs into SQL Queries

- Directly embedding user inputs into SQL queries is dangerous

```
public List<Permission> queryPermissionsByUserUnsafe(String user) {
  String sql = "SELECT perm FROM permissions WHERE username = '" + user + "'";
  List<Permission> perms = new ArrayList<>();
  Statement stmt = null;
  Connection conn = dataSource.getConnection();
  try {
    stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql);
    while (rs.next()) {
        String perm = rs.getString("perm");
        Permission permission = new Permission(perm);
        perms.add(permission);
    }
  } catch (SQLException e) {
    // etc
```

SQL Injection

- Code that directly embeds user inputs lays itself open to SQL injection attacks

```
String sql = "SELECT perm FROM permissions WHERE username = '" + user + "'";
```

  - What if the parameter came from user input and someone entered the following String?

```
"Bobby' OR '1'='1"
```

  - `sql` would become:

```
SELECT perm FROM permissions WHERE username = 'Bobby' OR '1' = '1'
```

- “Little Bobby Tables”
  - https://xkcd.com/327/

Preventing SQL Injection

- Preventing SQL injection is simple
  - **Never** create SQL by concatenating string input from the user
  - **Always** use a `PreparedStatement` to insert the values into the query
- **Important Note:** The parameter indices start with 1 (not 0)!

```
public List<Permission> queryPermissionsByUserSafe(String user) {
  String sql = "SELECT perm FROM permissions WHERE username = ?";
  List<Permission> perms = new ArrayList<>();
  PreparedStatement stmt = null;
  Connection conn = dataSource.getConnection();
  try {
    stmt = conn.prepareStatement(sql);
    stmt.setString(1, user);
    ResultSet rs = stmt.executeQuery();
    // etc
```

3. Perform a Query

- SQL calls are executed by a `PreparedStatement`
  - Use the Connection to prepare a `PreparedStatement`
  - Notice that there is no `;` at the end of the query string

```
public List<Department> queryDepartmentsByName(String name) {
  String sql = "SELECT deptno, dname, loc FROM dept WHERE dname = ?";
  List<Department> depts = new ArrayList<>();
  PreparedStatement stmt = null;
  Connection conn = dataSource.getConnection();
  try {
    stmt = conn.prepareStatement(sql);
    stmt.setString(1, dname);
    ResultSet rs = stmt.executeQuery();
    // process returned data
    ...
  } catch (SQLException e) {
    e.printStackTrace(); // better way coming soon
  } finally {
    ...  // IMPORTANT: close connection
  }
}
```

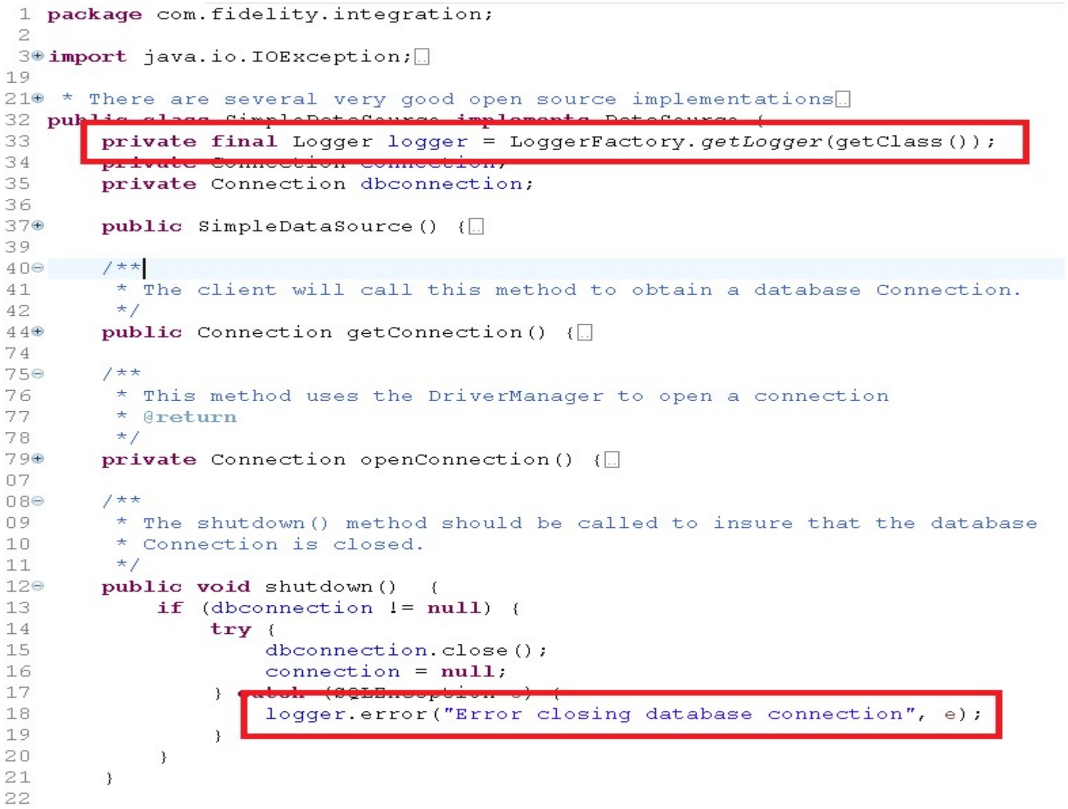
4. Parsing Results from SELECT Statements

- `rs.next()` moves to the next row of result set
- `rs.getInt()`, `rs.getString()`, etc. retrieve fields from current row
- This example requires a constructor with arguments for Department
- It is also a very good idea to define the `hashCode()` and `equals()` methods in the model

```
public List<Department> queryDepartmentsByName(String name) {
  String sql = "SELECT deptno, dname, loc FROM dept "
             + "WHERE dname = ?";
  List<Department> depts = new ArrayList<>();
  PreparedStatement stmt = null;
  Connection conn = dataSource.getConnection();
  try {
    stmt = conn.prepareStatement(sql);
    stmt.setString(1, name);
    ResultSet rs = stmt.executeQuery();
    while (rs.next()) {
      int deptNumber = rs.getInt("deptno");    // or rs.getInt(1)
      String deptName = rs.getString("dname"); // or rs.getString(2)
      String loc = rs.getString("loc");
      Department dept = new Department(deptNumber, deptName, loc);
      depts.add(dept);
    }
  } catch (SQLException e) {
    e.printStackTrace(); // better way coming soon
  } finally { ... /* close connection */ }
  return depts;
}
```

Logger

- Logger should be used for all error/catch paths //BRAD NOTE use a form of Logger; more details on Loggers in a later.



```
package com.fidelity.integration;

import java.io.IOException;□

 * There are several very good open source implementations□
public class SimpleDataSource implements DataSource {
    private final Logger logger = LoggerFactory.getLogger(getClass());
    private Connection connection;
    private Connection dbconnection;

    public SimpleDataSource() {□

    /**
     * The client will call this method to obtain a database Connection.
     */
    public Connection getConnection() {□

    /**
     * This method uses the DriverManager to open a connection
     * @return
     */
    private Connection openConnection() {□

    /**
     * The shutdown() method should be called to insure that the database
     * Connection is closed.
     */
    public void shutdown()  {
        if (dbconnection != null) {
            try {
                dbconnection.close();
                connection = null;
            } catch (SQLException e) {
                logger.error("Error closing database connection", e);
            }
        }
    }
```

5. Close JDBC Resources

- When done with a `ResultSet`, `Statement`, or `Connection`, close it
  - Closing a `Connection` automatically closes its `Statements` and `ResultSets`
  - But the JDBC driver never automatically closes a `Connection`
- You need to close every `Connection` to avoid resource leaks
  - Some databases (e.g., Oracle) keep connections open indefinitely unless explicitly closed
- `close()` may throw a checked exception—requires additional exception handling

```
try {
   …  // database operations
} finally {
   if (conn != null) {
      try {
         conn.close();
      } catch (SQLException e) {
         logger.error("can't close connection", e);
      }
   }
```

Closing the Connection

- Database Connections are precious resources
  - Only a limited number of Connections can be open at any moment
  - It may not be possible to connect to a database when all Connections are in use
- Opening and closing Connections are expensive operations
  - But keeping Connections open may prevent other users from connecting to the database
- The solution is to use a **DataSource**
  - Most enterprise DataSources define a pool of database Connections
  - Get a Connection by calling getConnection() on the DataSource
  - Return a Connection to the pool by calling close() on the Connection

Chapter Concepts

JDBC

Executing Queries

Implementing a Data Access Object

Handling NULL

Enumerated Types

Chapter Summary

Data Access Object

- Name: Data Access Object (DAO)
  - Not to be confused with DOA
- Problem: There are several to many places in your application that need to communicate with a data source such as a relational database
- Solution: Encapsulate the data source communication code in one object—the Data Access Object. Other parts of the program can call on the DAO to communicate with the data source
- Consequences:
  - The only part of the program that needs to know the data source details is the DAO
  - The rest of the program is insulated from any data source-specific details

DAO Implementation

- The DAO should get a Connection from a DataSource
  - The DAO should only use the Connection to prepare a PreparedStatement
  - The DAO should call close() on the Connection when finished communicating with the database
    - This returns the Connection to the DataSource
  - The DataSource will be responsible for closing the Connection with the database
- DAO methods should map to required database operations
  - How to hide how data objects are being created?
  - How to handle exceptions?

Exception Handling

- Many of the lines of code throw Exceptions
  - Almost all JDBC methods can throw a `SQLException`
    - For example, `rs.next()`, `rs.close()`, etc.

- Few database-driven applications can work if the database is inaccessible
  - Simply catch the exception and re-throw as a custom `RuntimeException`
    - The Business or Presentation layer should catch and deal with exception gracefully

```java
try {
  // all the database code
} catch (SQLException e) {
  logger.error("Cannot execute SQL query for dept: {}", sql, e);
  throw new DatabaseException("Cannot execute SQL query for dept: " + sql, e);
}
```

Note that the original exception is "chained"
so it appears in the stack trace as "caused by"

Without Try-With-Resources

- Simplifying the exception handling allows us to clean up our code with try-with-resources

```java
@Override
public List<Department> queryDepartmentsByName(String name) {
  String sql = "SELECT deptno, dname, loc FROM dept WHERE dname = ?";
  List<Department> depts = new ArrayList<>();
  PreparedStatement stmt = null;
  Connection conn = dataSource.getConnection();
  try {
    stmt = conn.prepareStatement(sql);
    ... // execute statement, process result set
  } catch (SQLException e) {
    logger.error("Cannot execute SQL query for dept: {}", name, e);
    throw new DatabaseException("Cannot execute SQL query for dept: " + name, e);
  } finally {
    if (conn != null) {
      try {
        conn.close();
      } catch (SQLException e) {
        logger.error("Cannot execute close connection", e);
      }
    }
    if (stmt != null) { ... /* similar code to close stmt */ }
  }
  return depts;
}
```

Have to declare `conn` outside the `try-catch-finally` so we can use it in `finally`

This `finally` block is particularly ugly. Exception handling in a `finally` block is always troublesome because we cannot easily throw an exception without masking any exception from the `catch` block and we cannot easily tell if the `catch` block threw an exception!

Using Try-With-Resources

- Any resource that implements `AutoCloseable` (or `Closeable`) can be used in a try-with-resources block
  – Resource is automatically closed without needing to write a `finally` block
  – Allows better scoping of the resource

```
@Override
public List<Department> queryDepartmentsByNameSimpler(String name) {
  String sql = "SELECT deptno, dname, loc FROM dept WHERE dname = ?";
  List<Department> depts = new ArrayList<>();
  try (Connection conn = dataSource.getConnection();
       PreparedStatement stmt = conn.prepareStatement(sql)) {
    stmt.setString(1, name);
    ... // execute statement, process result set
  } catch (SQLException e) {
    logger.error("Cannot execute SQL query for dept: {}", name, e);
    throw new DatabaseException("Cannot execute SQL query for dept: " + name, e);
  }
 return depts;
}
```

Initialize the resource here. Can have multiple lines with semicolons.

Scope of `conn` now restricted to the `try` block

No ugly `finally` block. Connection and statement are automatically closed.

Logging with SLF4J

- It is often useful to know what is happening in a running production system
  - For troubleshooting; to choose optimizations; to allow detailed application analysis
  - Java logging makes this easy
  - SLF4J: flexible, easy-to-use logging framework
    - https://www.slf4j.org/apidocs/index.html
- Log parameters
  - Rather than this:

```
System.out.println("Compute iteration " + k);
```

  - Use this:

```
logger.debug("Compute iteration {}", k);
```

  - Parameters are not evaluated unless the appropriate logging level is enabled
- SLF4J tutorial: https://www.baeldung.com/slf4j-with-log4j2-logback
- **Log4j2 vulnerability**: https://cve.mitre.org/cgi-bin/cvename.cgi?name=2021-44228

Using Java Logging

- Log exceptions with `logger.error()`
  – By default, includes the stack trace in the log file

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class EmployeeDao {
    private final Logger logger = LoggerFactory.getLogger(getClass());
    ...
    try {
        ...
    } catch (SQLException ex) {
        logger.error("Cannot execute SQL query for dept {}", name, ex);
        ...
```

`logger.error()` logs stack of exception

Dates and Times — Simple Cases (No Time Zone)

- Since Java 8, the preferred representations are from `java.time`
  - `LocalDate` represents a date without time or time zone
  - `LocalTime` represents a time without time zone
  - `LocalDateTime` represents a date and time without time zone
- JDBC provides simple mappings

Java	JDBC (ANSI SQL)	Oracle	Comments
LocalDate	DATE	DATE	Oracle DATE includes time, stripped out by JDBC
LocalTime	TIME	DATE TIMESTAMP	Oracle DATE includes date, stripped out by JDBC
LocalDateTime	TIMESTAMP	DATE TIMESTAMP	TIMESTAMP contains fractional seconds, Oracle DATE does not

Querying Dates and Times

```
CREATE TABLE datetimetest (
    date_test          DATE,
    time_test          DATE,
    datetime_test      DATE,
    timestamp_test     TIMESTA
)
```

```
String sql = "SELECT * FROM datetimetest";//BRAD NOTE * will not work
try (PreparedStatement stmt = conn.prepareStatement(sql)) {
    ResultSet rs = stmt.executeQuery();
    while (rs.next()) {
        LocalDate ld = rs.getDate("date_test").toLocalDate();
        System.out.println(ld);

        LocalTime lt = rs.getTime("time_test").toLocalTime();
        System.out.println(lt);

        LocalDateTime ldt1 = rs.getTimestamp("datetime_test").toLocalDateTime();
        System.out.println(ldt1);

        LocalDateTime ldt2 = rs.getTimestamp("timestamp_test").toLocalDateTime();
        System.out.println(ldt2);
    }
}
```

```
2017-12-31
23:59:59
2017-12-31T23:59:59
2017-12-31T23:59:59.123456
```

Inserting Dates and Times

```
CREATE TABLE datetimetest (
    date_test         DATE,
    time_test         DATE,
    datetime_test     DATE,
    timestamp_test    TIMESTAMP
)
```

```
String sql = "INSERT INTO datetimetest VALUES (?, ?, ?, ?)";
try (PreparedStatement stmt = conn.prepareStatement(sql)) {
    LocalDate ld = LocalDate.of(2017, 12, 31);
    stmt.setDate(1, Date.valueOf(ld));

    LocalTime lt = LocalTime.of(23, 59, 59);
    stmt.setTime(2, Time.valueOf(lt));

    LocalDateTime ldt1 = LocalDateTime.of(ld, lt);
    stmt.setTimestamp(3, Timestamp.valueOf(ldt1));

    LocalDateTime ldt2 = LocalDateTime.of(2017, 12, 31, 23, 59, 59, 123456000);
    stmt.setTimestamp(4, Timestamp.valueOf(ldt2));

    stmt.executeUpdate();
}
```

The JDBC types are `java.sql.Date`, `java.sql.Time` and `java.sql.Timestamp`

ROI TRAINING
MAXIMIZE YOUR TRAINING INVESTMENT

Fidelity LEAP
Technology Immersion Program

Working with `BigDecimal` in JDBC

- Typical insert or update

```java
stmt.setLong(1, employee.getId());
stmt.setString(2, employee.getName());
stmt.setBigDecimal(3, employee.getSalary());
stmt.executeUpdate();
```

- Typical select

```java
stmt.setLong(1, department);
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
    Employee employee = new Employee(rs.getLong("EMPNO"),
                                     rs.getString("ENAME"),
                                     rs.getBigDecimal("SAL"));
    employees.add(employee);
}
```

What to Test — `SELECT`

- Need to verify that a database operation succeeded
  – Usually, a DAO method converts a result set into one or more objects
  – Test goals: Did the query create valid objects? Were the right objects created?

```
SELECT … FROM dept … ORDER BY deptno
```

Add ORDER BY so results are in a predictable order

```
private Department dept10 = new Department(10, "ACCOUNTING", "NEW YORK");
private Department dept40 = new Department(40, "OPERATIONS", "BOSTON");

@Test
public void testQueryAllDepartments() {

  List<Department> depts = dao.queryAllDepartments();

  assertEquals(4, depts.size());
  assertEquals(dept10, depts.get(0)); // verify first item
  assertEquals(dept40, depts.get(depts.size() - 1)); // verify last item
}
```

```
public class Department {
  public boolean equals(Object o) { … }
  public String toString() { … }
  …
}
```

`assertEquals()` uses `equals()` for comparisons and `toString()` for error messages

Exercise 7.2: Creating Objects from Database Query

Do Now

60 min

- Complete this exercise described in the Exercise Manual

- Use JDBC—make sure you complete all the steps to get data from the database
- Use TDD—how will you test your objects are created correctly?
- Use Eclipse for debugging
  - Starting at the top of a stack trace, find your own code within the stack trace and click it to take you to a location very close to the error
  - Read the entire error message to understand what went wrong

//BRAD NOTE: lecture balance of the chapter, then do lab 7.2

Chapter Concepts

JDBC

Executing Queries

Implementing a Data Access Object

Handling NULL

Enumerated Types

Chapter Summary

Reading Database `NULL` in JDBC

- JDBC's handling of NULL columns is not always intuitive
- Example: In PRODUCT table, SHIPPING_WEIGHT is a nullable NUMERIC column
- `Product` class defines a nullable `shippingWeight` property:

```
public class Product {
    private Double shippingWeight;
    public void setShippingWeight(Double weight) { shippingWeight = weight; }
```

Properties of type `Double` can be null

- Task: read SHIPPING_WEIGHT from DB and set a `Product`'s `shippingWeight` property
  - But if column value is NULL, set `shippingWeight` to `null`
  - First attempt:

```
Double weight = rs.getDouble("shipping_weight");
product.setShippingWeight(weight);
```

Doesn't work! If column is `NULL`, `getDouble()` returns `0.0`

Reading Database NULL in JDBC (continued)

- For Java primitives, NULL maps to 0 for numeric types, false for boolean
  - To find out if a column really is NULL, use rs.wasNull() *after* getting the column
  - To allow null values, replace primitive fields with wrapper classes (Double, Boolean)

```
Double weight = rs.getDouble("shipping_weight");
if (rs.wasNull()) {
   weight = null;
}
product.setShippingWeight(weight);
```

- For objects (String, Date, BigDecimal, etc.), JDBC maps database NULL to Java null
  - Test value before calling conversion methods to avoid NullPointerException

```
LocalDate hireDate = null;
Date dbHireDate = rs.getDate("hiredate");
if (dbHireDate != null) {
   hireDate = dbHireDate.toLocalDate();
}
employee.setHireDate(hireDate);
```

Don't call toLocalDate() if hiredate column was NULL

Writing Database `NULL` in JDBC

- `Statement` set methods that accept object types interpret Java `null` as database `NULL`

```
stmt.setDate(5, null);
```

Okay

- `Statement` has special methods that accept Java primitives and set a column to `NULL`

```
if (employee.getComm() == null) {
  stmt.setNull(7, java.sql.Types.NUMERIC);
} else {
  stmt.setDouble(7, employee.getComm());
}
```

This is the JDBC type (based on ANSI SQL) that maps to Oracle `NUMBER`

- If a getter method might return `null`, test the value before setting a column

```
stmt.setDouble(7, employee.getComm());
```

Generates an NPE if `getComm()` returns null

Chapter Concepts

JDBC

Executing Queries

Implementing a Data Access Object

Handling NULL

Enumerated Types

Chapter Summary

Handling `enum`

- To store an `enum` property in a database table, you could store the `enum`'s string value
  - For String/VARCHAR2 columns, just use the `enum` name
  - Use standard methods `toString()` and `valueOf()`

```
public enum PerformanceReviewResult {
  BELOW,
  AVERAGE,
  ABOVE
}
```

```
public class Employee {
  private PerformanceReviewResult review;
  public PerformanceReviewResult getPerformanceReviewResult() {
    return review;
  }
```

```
String perfRev = rs.getString("perf_rev_name"); // "BELOW", "AVERAGE", "ABOVE"
PerformanceReviewResult revResult = PerformanceReviewResult.valueOf(perfRev);
```

```
PerformanceReviewResult review = employee.getPerformanceReviewResult();
stmt.setString(3, review.toString()); // "BELOW", "AVERAGE", "ABOVE"
```

Handling enum (continued)

- More commonly, `enums` are stored in the database as numeric values
  – `enum` needs constructor, static factory method, and getter method

```
public enum PerformanceReviewResult {
  BELOW(1), AVERAGE(3), ABOVE(5);
  private int code;
  private PerformanceReviewResult(int code) {
    this.code = code;
  }
  public static PerformanceReviewResult of(int code) {
    for (PerformanceReviewResult revRes :
             PerformanceReviewResult.values()) {
        if (revRes.getCode() == code) {
            return revRes;
        }
    }
    throw new IllegalArgumentException("bad code: " + code);
  }
  public int getCode() { return code; }
```

Call `enum` constructor with integer argument

Private constructor

Static factory method converts integer to `enum` value

```
int perfRev = rs.getInt("perf_rev_code");
PerformanceReviewResult revResult =
    PerformanceReviewResult.of(perfRev);
```

```
PerformanceReviewResult review =
    employee.getPerformanceReviewResult();
stmt.setInt(3, review.getCode()); // 1, 3, 5
```

Chapter Concepts

JDBC

Executing Queries

Implementing a Data Access Object

Handling NULL

Enumerated Types

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- How Java provides a set of interfaces for connecting to and working with SQL databases
  - Database vendors provide implementations of those interfaces
  - Allows you to write reusable database code
- Why working with databases requires a standard set of steps
  - You will practice these steps and then build them into an object
- Why database security is essential
  - Guard against bad user input
  - SQL injection attacks
- Data Access Object Design Pattern
  - What is the problem we need to solve?
  - What does the implementation look like?
  - What are the trade-offs?

Key Points

- Java interfaces provide standard, portable way to access SQL databases
- Accessing databases provides an effective way to create business objects
  - Code for working with databases should be encapsulated
- Never concatenate user inputs directly to SQL queries
  - You cannot trust data directly given by users
  - Use `PreparedStatement` to avoid SQL injection attacks
- Create a data access object (DAO) that rest of code base will use
  - DAO consolidates all database access in a single class
  - Receives a Connection from a DataSource in each method
  - Methods of DAO map to database operations